

ABSZTRAKCIÓ ADATOKKAL

Felhasználói adattípusok: ismét a datatype deklarációról

- person néven új összetett típust hozunk létre:

```
datatype person = King
                | Peer of string * string * int
                | Knight of string
                | Peasant of string
```

- Az új típusnak négy *adatkonstruktor*a (röviden: *konstruktor*) van: King, Peer, Knight és Peasant.
- King ún. *adatkonstruktorállandó*, a többi ún. *adatkonstruktorfüggvény*.
- Az adatkonstruktoroknak is van típusuk:

```
King :      person
Peer  :      string * string * int -> person
Knight :      string -> person
Peasant :      string -> person
```

A datatype deklaráció (folyt.)

```
King :    person
Peer  :    string * string * int -> person
Knight :  string -> person
Peasant : string -> person
```

- King (király) csak egy van, ezért definiálhattuk konstruktorállandóként.
- A Peer-t (főnemest) nemesi címe (`string`), birtokának neve (`string`) és sorszáma (`int`) azonosítja.
- A Knight-ot (lovagot) és a Peasant-ot (parasztot) csupán a neve (`string`) azonosítja.
- Példa a `person` adattípus alkalmazására:

```
val persons = [King, Peasant "Jack Cade", Knight "Gawain",
               Peer("Duke", "Norfolk", 9)];
```

```
> val persons = [King, Peasant "Jack Cade", ...] : person list
```

- Az egyes esetek mintaillesztéssel választhatók szét.
- Minden esetet le kell fedni mintával; ha nem, figyelmeztetést kapunk.
- A minták tetszőlegesen összetettek lehetnek.

A datatype deklaráció (folyt.)

- Az alábbi példában a négy közül az egyik a Peasant name *minta*, és benne name a *mintaazonosító*.

```
(* title : person -> string
   title p = p megszólítása *)
fun title King = "His Majesty the King "
  | title (Peer (deg, ter, _)) = "The " ^ deg ^ " of " ^ ter
  | title (Knight name) = "Sir " ^ name
  | title (Peasant name) = name
```

- A sirs függvény az összes Knight nevét összegyűjti a person típusú személyek egy listájából (a változatok sorrendje *fontos* az _ miatt!):

```
(* sirs : person list -> string list
   sirs ps = az összes Knight nevének listája *)
fun sirs [] = []
  | sirs ((Knight s)::ps) = s::sirs ps
  | sirs (_::ps) = sirs ps
```

A datatype deklaráció (folyt.)

- Ha más lenne a változatok sorrendje, a `_ :: ps` minta nemcsak a `King`-re, a `Peer`-re és a `Peasant`-ra illeszkedne (ti. ezek helyett áll a példában), hanem a `Knight`-ra is.
- Az összes diszjunkt eset felsorolása segíti az algoritmus helyességének belátását, bizonyítását.
- Azért vontunk össze három esetet egyetlen változatban, mert a részletezésük hosszabbá tenné a program szövegét is, végrehajtását is.
- A bizonyítás nem okoz gondot, ha a függvény harmadik sorát (`sirs (_ :: ps) = sirs ps`) *feltételes egyenletnek* tekintjük:

$$\text{sirs}(p :: ps) = \text{sirs } ps \text{ if } \forall s. p \neq \text{Knight } s.$$

A datatype deklaráció (folyt.)

- A sorrend még fontosabb a következő példában, amelyben személyek hierarchiáját vizsgáljuk. Itt 16 helyett csak 7 esetet kell megkülönböztetnünk: azokat, amelyek *igaz* eredményt adnak.

```
(* superior : person * person -> bool
   superior (p, r)= igaz, ha p magasabb rangú r-nél *)
fun superior (King, Peer _) = true
  | superior (King, Knight _) = true
  | superior (King, Peasant _) = true
  | superior (Peer _, Knight _) = true
  | superior (Peer _, Peasant _) = true
  | superior (Knight _, Peasant _) = true
  | superior _ = false
```

Felsorolós típus `datatype` deklarációval

- Gyakori, hogy egy név csak néhány különböző értéket vehet fel (azaz a név által felvehető értékek halmaza kis számosságú), ilyen esetben érdemes *felsorolós típust* létrehozni a `datatype` deklarációval. Pl.

```
datatype degree = Duke | Marquis | Earl | Viscount | Baron
```

- A felsorolós típusnak csak *konstruktorállandói* vannak. Az új típus alkalmazásához a `person` típust újra deklarálnunk kell:

```
datatype person = King
                | Peer of degree * string * int
                | Knight of string
                | Peasant of string
```

Felsorolásos típus `datatype` deklarációval (folyt.)

- A `degree` típusú adatok feldolgozásakor külön-külön elemezzük az előforduló eseteket, pl.

```
(* lady : degree -> string
   lady p = p főnemes hitvesének rangja *)
fun lady Duke      = "Duchess "
  | lady Marquis   = "Marchioness "
  | lady Earl      = "Countess "
  | lady Viscount  = "Viscountess "
  | lady Baron     = "Baroness "
```

- A belső `bool` típushoz hasonló `Bool` típust és hozzá a `Not` függvényt például így is deklarálnánk, ill. definiálnánk:

```
datatype Bool = True | False
(* Not : Bool -> Bool
   Not b = b negáltja *)
fun Not True = False | Not False = True
```


Polimorf adattípusok

- Láttuk, hogy a `list postfix` pozíciójú *típusoperátor*, nem típus: a `datatype` deklaráció az adatkonstruktorok mellett *típuskonstruktort* is létrehoz.
- A belső `'a list` típushoz hasonló `'a List` listát és vele együtt a `Nil` és a `Cons` *adatkonstruktorokat* például így definiálhatjuk:

```
datatype 'a List = Nil | Cons of 'a * 'a List;
```

- A `Cons` *adatkonstruktorfüggvény* alkalmazásával elég körülményes a listák létrehozása. Az 1, 2, 3, 4 sorozatot például így kell megadni:

```
Cons(1, Cons(2, Cons(3, Cons(4, Nil))));
```

- Bevezethetjük az *infix* pozíciójú `:::` *adatkonstruktoroperátort*:

```
infix 5 ::: ; val op ::: = Cons
```

- Az *infix hatospontot* magában a típusdeklarációban is definiálhatjuk:

```
infix 5 ::: ; datatype 'a List = Nil | ::: of 'a * 'a List
```

Polimorf adattípusok: megkülönböztetett egyesítés

- Következő példánk két típus *megkülönböztetett egyesítése*, más néven diszjunkt uniója:

```
datatype ('a, 'b) disun = In1 of 'a | In2 of 'b
```

- Itt három dolgot definiáltunk:

1. a kétargumentumú `disun` típusoperátort,
2. az `In1` : `'a -> ('a, 'b) disun` és
3. az `In2` : `'b -> ('a, 'b) disun` adatkonstruktorfüggvényeket.

- `('a, 'b) disun` az `'a` és `'b` típusok megkülönböztetett egyesítése. *Megkülönböztetettnek* nevezzük az egyesítést, mert később is bármikor meg tudjuk mondani, hogy egy `('a, 'b) disun` típusú pár egyik vagy másik eleme melyik alaptípusból származik. Az új típusba tartozó értékek `In1 x` alakúak, ha `x 'a` típusú, és `In2 y` alakúak, ha `y 'b` típusú.
- Az `In1` és `In2` konstruktorfüggvények olyan *címkének* tekinthetők, amelyek az `'a` típust megkülönböztetik a `'b` típustól.

Megkülönböztetett egyesítés (folyt.)

- A megkülönböztetett egyesítés lehetővé teszi, hogy különböző típusokat használjunk ott, ahol egyébként csak egyetlen típust használhatnánk (vö. objektum-orientált programozás, ahol pl. egy *alakzat* osztálynak *téglalap*, *háromszög* vagy *kör* nevű leszármazottai lehetnek).
- Az SML-ben megkülönböztetett egyesítéssel tudunk létrehozni *különböző típusú elemekből* álló listát:

```
[In2 King, In1 "Skócia"] : ((string, person) disun) list;
[In1 "zsarnok", In2 1040] : ((string, int) disun) list
```

- A lehetséges eseteket most is *mintaillesztéssel* elemezhetjük, pl.

```
(* concat : (string, 'a) disun list -> string
   concat d = a d diszjunkt unió In1 címkéjű
               elemeinek konkatenációja *)
fun concat [] = ""
  | concat (In1 s :: ls) = s ^ concat ls
  | concat (In2 _ :: ls) = concat ls;
```

Megkülönböztetett egyesítés (folyt.)

- Egy példa concat alkalmazására:

```
concat [In1 "Ó! ", In2 King, In1 "Skócia"];
```

```
> val it = "Ó! Skócia" : string
```

- Az In1 konstruktorfüggvény típusa 'a -> ('a, 'b) disun, ezért a string típusú "Ó! " argumentumra alkalmazva (string, 'b) disun típusú érték az eredmény.
- Az In2 konstruktorfüggvény típusa 'b -> ('a, 'b) disun, ezért a person típusú King kifejezésre alkalmazva ('a, person) disun típusú érték az eredmény.
- Az [In1 "Ó!", In2 King, In1 "Skócia"] kifejezésben mindkét alaptípust lekötjük, ezért ennek a listának a típusa: ((string, person) disun) list.
- Az [In2 "Ó", In2 King, In1 "Skócia"] kifejezés kiértékelése hibajelzést eredményez, mert a 'b típusváltozót nem lehet ugyanabban a kifejezésben egyszer így, másszor úgy lekötni.

LOKÁLIS DEKLARÁCIÓ, EGYIDEJŰ DEKLARÁCIÓ

Deklaráció lokális érvényű deklarációval: `local`-deklaráció

- Már megismertük a lokális deklarációt használó *kifejezést*.
- Lokális deklarációt használó *deklarációt* használunk, ha egyes deklarációkat fel akarunk használni más deklarációkban, miközben *el akarjuk rejtetni* őket a program többi része előtt.
- Szintaxisa:


```
local d1      ahol d1 egy nemüres deklarációsorozat,
in d2                d2 egy másik nemüres deklarációsorozat.
end
```

- Példa:

```
(* length : 'a list -> int
   length zs = a zs lista hossza *)
local
  (* len : 'a list * int -> int
     len (zs, n) = az n és a zs lista hosszának összege *)
  fun len ([], n)      = n
    | len (_::zs, n) = len(zs, n+1)
in
  fun length zs = len(zs, 0)
end
```

Egyidejű deklaráció

- Típusok, ill. értékek *egyidejűleg* is deklaráálhatók az `and` kulcsszó alkalmazásával (az utóbbira már láttunk példákat).
- Vegyük a következő deklarációsorozatokat:

```
type sor = int; type osz = int;
datatype fa = L | B of fa * fa;
    datatype 'a verem = > | | >> of 'a * 'a verem;
val v1 = "a"; val v2 = "z";
fun f1 i = i + 1; fun f2 i = i - 1;
```

A fenti deklarációkat az SML-értelmező a *megadott sorrendben* értékeli ki.

```
type sor = int and osz = int;
datatype fa = L | B of fa * fa and
    'a verem = > | | >> of 'a * 'a verem;
val v1 = "a" and v2 = "z";
fun f1 i = i + 1 and f2 i = i - 1;
```

Az `and` szócskával elválasztott deklarációkat az SML-értelmező *egyidejűleg* értékeli ki.

Egyidejű deklaráció (folyt.)

- Egyidejű deklarációt kell használnunk kölcsönösen rekurzív függvények definiálására. Példa:

```
fun even 0 = true | even n = odd(n-1)
and odd 0 = false | odd n = even(n-1);
```

- Egyidejű deklarációt használhatunk két vagy több kötés egyidejű felcserélésére. Példa:

```
val v1 = "a"; val v2 = "z"; val v1 = v2 and v2 = v1;
```

- Egyidejű deklarációt használhatunk, ha fölülről lefelé haladva akarunk programot írni. Példa:

```
fun length zs = len zs 0
and len [] i = i | len (_ :: xs) i = len xs (i+1);
```

- A polimorf függvényeket a szekvenciális és az egyidejű deklaráció eltérően kezeli, mivel a típuslevezetést az SML-értelmező a teljes kifejezésre alkalmazza. Példa:

```
fun id x = x; fun hi () = id 3; fun nr () = id 4.0;
fun id x = x and hi () = id 3 and nr () = id 4.0;
```

Az első sor kiértékelésekor `id 'a -> 'a` típusú. A második sor kiértékelésekor `id int -> int` és `real -> real` típusú lenne egyszerre, ami lehetetlen.

ESETSZÉTVÁLASZTÁS, OPCIONÁLIS ÉRTÉK

Esetszétválasztás (case)

```
case E of P1 => E1 | P2 => E2 | ... | Pn => En
```

Az SML-értelmező – balról jobbra és fölülről lefelé haladva – megpróbálja E-t P1-re illeszteni, ha nem sikerül, P2-re s.í.t. A case-kifejezés eredménye az E kifejezésre illeszkedő első P_i mintához tartozó E_i kifejezés lesz.

A case is csak szintaktikus édesítőszert, ui. helyettesíthető fn-jelöléssel:

```
(fn P1 => E1 | P2 => E2 | ... | Pn => En) E
```

Például a lady függvényt így is definiálhattuk volna:

<pre>datatype degree = Duke Marquis Earl Viscount Baron (* lady : degree -> string lady p = p főnemes hitvesének rangja *) fun lady p = case p of Duke => "Duchess " Marquis => "Marchioness" Earl => "Countess" Viscount => "Viscountess" Baron => "Baroness"</pre>	<pre>(* lady : degree -> string lady p = p főnemes hitvesének rangja *) fun lady p = (fn Duke => "Duchess " Marquis => "Marchioness" Earl => "Countess" Viscount => "Viscountess" Baron => "Baroness") p</pre>
---	--

Opcionális érték kezelése ('a option)

```
datatype 'a option = NONE | SOME of 'a
```

Függvények az Option könyvtárból:

```
val getOpt      : 'a option * 'a -> 'a
val isSome     : 'a option -> bool
val valOf      : 'a option -> 'a
val filter     : ('a -> bool) -> 'a -> 'a option
val map        : ('a -> 'b) -> 'a option -> 'b option
val mapPartial : ('a -> 'b option) -> ('a option -> 'b option)
```

getOpt (xopt, d) = x if xopt is SOME x, d otherwise.

isSome xopt = true if xopt is SOME x, false otherwise.

valOf xopt = x if xopt is SOME x, raises Option otherwise.

filter p x = SOME x if p x is true, NONE otherwise.

map f xopt = SOME(f x) if xopt is SOME x, NONE otherwise.

mapPartial f xopt = f x if xopt is SOME x, NONE otherwise.

Példák opcionális értékek kezelésére

- Egészlista legnagyobb elemének kiválasztása

Üres listának nincs legnagyobb eleme; egyelemű lista egyetlen eleme a „legnagyobb”; legalább kételemű lista legnagyobb eleme az első elem és a maradéklista elemei közül a legnagyobb.

```
(* maxl : int list -> int option
   maxl ns = az ns egészlista legnagyobb eleme *)
fun maxl []      = NONE      (* üres *)
  | maxl [n]     = SOME n    (* egyelemű *)
  | maxl (n::ns) =           (* legalább kételemű *)
      SOME(Int.max(n, valOf(maxl ns)))
```

- Füzér elején álló karaktersorozat átalakítása egész számmá

```
val Int.fromString : string -> int option (* Overflow *)
```

```
Int.fromString s = SOME i if a decimal integer numeral can be scanned
  from a prefix of string s, ignoring any initial whitespace;
  NONE otherwise. A decimal integer numeral, after any initial
  whitespace, must have the form: [+~-]?[0-9]+
```

```
Int.fromString "1234"; Int.fromString "-1234"; Int.fromString "~1234";
Int.fromString "+1234"; Int.fromString "+007"; Int.fromString "alma"
```

AZ ORDER TÍPUS



Az order típus

Az order típus definíciója (ld. `General.sig`)

```
datatype order = LESS | EQUAL | GREATER
```

[order] is used as the return type of comparison functions.

Példák az SML-alapkönyvtárból (SML Basis Library)

```
Int.compare      : int * int -> order
Char.compare    : char * char -> order
Real.compare    : real * real -> order
String.compare  : string * string -> order
Time.compare    : time * time -> order
```

KIVÉTELKEZELÉS



Kivételkezelés

- Kivételt az `exception` kulcsszóval deklarálunk, a `raise` kulcsszóval jelzünk, a `handle` kulcsszóval bevezetett kifejezésben kezelünk.
- A kivételt általában hibák jelzésére használjuk, de használhatjuk visszalépés kezelésére is (az utóbbira példa a `valtas` függvényben látható a következő fóliák egyikén).
- A kivételdeklaráció az adattípus-deklarációra (`datatype`-deklarációra) emelkedtet:
`exception name; exception name of ty.`
- Példák kivétel deklarálására: `exception Valt; exception Hiba of char * int.`
- A kivételkonstruktor állandó vagy függvény lehet. Példák: `Valt : exn, Hiba : char * int -> exn.`
- A kivételdeklaráció speciális adattípus-deklaráció, ui. az utóbbival ellentétben dinamikusan *bővíti* a kivételkonstruktorok halmazát.
- Kivétel jelzésére a `raise` kulcsszóval kezdődő speciális kifejezést kell használnunk.
- Példák kivétel jelzésére: `raise Valt, raise Hiba("#N", 4).`
- `raise` (hipotetikus) típusa: `exn -> 'a.`

Kivételkezelés (folyt.)

- `raise` alkalmazásának eredménye az ún. *kivételcsomag*. Mivel a kivételcsomag polimorf típusú, bármely más típusal kompatibilis.
- A kivétel kezelése a `case`-szerkezetre emlékeztet: `E handle P1 => E1 | ... | Pn => En`
- Ha `E` „közönséges” értéket ad eredményül, a kivételkezelő egyszerűen továbbadja az eredményt.
- Ha `E` eredménye *kivételcsomag*, az SML megpróbálja illeszteni a `P1`, ..., `Pn` mintákra.
 - Ha `Pi` ($1 \leq i \leq n$) az első illeszkedő minta, akkor `Ei` a kivételkezelő eredménye.
 - Ha egyetlen minta sem illeszkedik a kivételcsomagra, a kivételkezelő továbbpasszolja.
- Példák kivétel kezelésére:
 - `erme :: váltas (erme::ermelista) (osszeg-erme)`
`handle Valt => váltas ermelista osszeg`
 - `(fn i => kivKez i handle Hiba(c, i) => (print(str c); i-1)) 0`
- `handle` (hipotetikus) típusa: `exn -> 'a`.
- Legyen `Ex` `exn` típusú kivétel, `e` pedig tetszőleges kifejezés; ekkor az `e handle Ex => c` (kivételkezelőt tartalmazó) kifejezésben `c`-nek `e`-vel azonos típusúnak kell lennie.

Kivételkezelés (folyt.)

- A következő programrészlet példa kivétel deklarálására, jelzésére és kezelésére

```
exception Hiba of char * int;
```

```
fun kivKez 0 = raise Hiba("#N", 4)  
  | kivKez ~9 = raise Hiba("#M", 9)  
  | kivKez n = n;
```

```
fun kivKezel i =  
    kivKez i handle Hiba("#N", i) => (print "N"; i)  
                | Hiba("#M", i) => (print "M"; i-1);
```

```
kivKezel 0 = 4;
```

```
kivKezel ~9 = 8;
```

```
kivKezel 7 = 7;
```

Kivételkezelés (folyt.)

● Példa visszalépés programozására kivételkezeléssel

```

exception Valt;

(* váltás : int list -> int -> int list
   váltás ermelista osszeg = a lehető legkevesebb érmét tartalmazó olyan
                           érmelista, amely elemeinek összege osszeg
   PRE : ermelista = a váltásra használható érmék csökkenő értéksorrendben
         osszeg >= 0
*)
fun váltás _ 0 = []
  | váltás [] _ = raise Valt
  | váltás (erme::ermelista) osszeg =
    if (* ha az adott érme túl nagy, a következővel próbálkozunk *)
       erme > osszeg then váltás ermelista osszeg
    (* ha az adott érmétől kezdve sikerül felváltani, az jó;
       ha nem, a következő érmével kezdjük újra az adott ponttól *)
    else erme :: váltás (erme::ermelista) (osszeg-erme)
        handle Valt => váltás ermelista osszeg;

váltás [50, 20, 10, 5, 2] 197 = [50, 50, 50, 20, 20, 5, 2];

```

Kivételkezelés (folyt.)

- A leggyakoribb belső kivételek

Név	Művelet, amely a kivételt kiválthatja
Bind	Értékdeklarációban a jobb oldali kifejezés nem illeszkedik a bal oldali mintára.
Chr	<code>chr pred succ</code>
Div	<code>/ div mod</code>
Domain	Az érték kilóg az értelmezési tartományból.
Empty	<code>hd tl last</code>
Fail	<code>compile load loadOne</code> <code>Fail : string -> exn</code>
Interrupt	Megszakítás <code>ctrl/c</code> -vel.
Io	Ki/beviteli hiba. <code>Io : {cause : exn, function : string, name : string}</code>
Match	Mintaillesztési hiba <code>case</code> és <code>handle</code> kifejezésben, vagy függvényalkalmazásban.
Option	Hiba egy <code>Option</code> könyvtárbeli függvény alkalmazásakor.
Overflow	<code>~ + - * / div mod abs ceil floor round trunc</code>
Size	<code>^ array concat fromList implode tabulate translate vector</code>
Subscript	<code>copy drop extract nth sub substring take update</code>

- `Fail` és `Io` kivételkonstruktorfüggvények, a többi `exn` típusú kivételkonstruktorállandó.
- `Option` csak `Option.Option` néven használható, ha nem nyitjuk meg az `Option` könyvtárat.

VISSZALÉPÉSES PROGRAMOZÁS



n vezér a sakktáblán

Hányféleképpen rakható n vezér egy $n * n$ méretű sakktáblára úgy, hogy ne üssék egymást?

- A sakktáblát egy olyan n hosszú sorvektorral írjuk le, amelynek egy-egy mezőjébe írt s szám a sakktábla egy-egy oszlopába lerakott vezér sorának a sorszáma ($0 \leq s < n$).
- Példa $n=4$ esetén:

```

+---+---+---+---+
| 2 | 0 | 3 | 1 |
+---+---+---+---+
0   ----> n-1

+---+---+---+---+
0 |   | q |   |   |
+---+---+---+---+
| |   |   |   | q |
+---+---+---+---+
V | q |   |   |   |
+---+---+---+---+
n-1 |   |   | q |   |
+---+---+---+---+

```

- A sorvektort listával valósítjuk meg.
- Egy listához *balról könnyű* új elemeket fűzni, ezért a táblát és a vezérek helyzetét leíró listát függőlegesen tükrözzük.

```

...-+---+---+---+
      | 3 | 0 | 2 |
...-+---+---+---+
n-1  <----  0

...-+---+---+---+
0      |   | q |   |
...-+---+---+---+
|   |   |   |   |
V   |   |   |   | q |
...-+---+---+---+
n-1   | q |   |   |
...-+---+---+---+

```

- Egy n hosszú sorvektor i -edik eleme az $n - (i + 1)$ -edik elem a listában.

n vezér a sakktáblán (folyt.)

Azt, hogy az új vezért ütik-e más vezérek a táblán, a *sorvektor* vizsgálatával dönthetjük el: az egyes elemek sorszama által meghatározott oszlopban az értékük által meghatározott sorban vezér van. A lerakás feltételei a következők:

1. Az új vezér nem kerülhet egy sorba egyetlen más vezérrel sem, azaz az új listalem *értéke* nem fordulhat elő a lista már felépített részében.
2. Az új vezér átlós irányban sem lehet egy vonalban más vezérrel. Ez azt jelenti, hogy ha a lista elejére (a 0. elemébe!) az *s* sorindexet akarjuk írni, akkor az *i*-edik listaelem értéke, feltéve, hogy van ilyen elem, nem lehet $s-i$, sem $s+i$.

A következő példa megvilágítja az esetet.

Ha a tábla 1. sorába akarjuk lerakni az új vezért, akkor az *x*-szel megjelölt mezőket kell megvizsgálnunk. Az új mezővel együtt a listának már három eleme van. Az 1-es indexű elem nem lehet $s-1$, sem $s+1$, a 2-es indexű elem pedig nem lehet $s-2$, sem $s+2$.

```

...+-----+-----+
      1 |   |   |   |
...+-----+-----+

      n-1 <--- 1   0
...+-----+-----+
0      |   | x |   |
...+-----+-----+
1      | q |   |   |
...+-----+-----+
|      |   | x |   |
V ...+-----+-----+
n-1    |   |   | x |
...+-----+-----+

```

A lista rekurzív algoritmussal dolgozható fel.

n vezér a sakktáblán: „ütésben van”-vizsgálat

```

(* utesbenVan : int list -> bool
    utesbenVan zs = igaz, ha a (hd zs) vezért legalább egy
                    (tl zs)-beli vezér üti
*)
fun utesbenVan [] = false
  | utesbenVan (z::zs) =
    let (* uV : int -> int -> int list -> bool
        uV s1 s2 rs = igaz, ha a z vezért legalább egy
                    rs-beli vezér üti
        *)
        fun uV _ _ [] = false
          | uV s1 s2 (r::rs) = z = r orelse
                               s1 = r orelse
                               s2 = r orelse
                               uV (s1-1) (s2+1) rs
    in
        uV (z-1) (z+1) zs
    end

```


n vezér a sakktáblán: egy megoldás előállítás

```

exception Zsakutca

(* vezerek0 : int -> int list
   vezerek0 n = a feladvány egy megoldása n vezér esetén
*)
fun vezerek0 n =
  let (* vez : int -> int list -> int list
       vez z zs = egy megoldás n vezér esetén *)
      fun vez z zs =
        if (* vissza kell lépni, ha z=0 és ütésben van *)
           z = 0 andalso utesbenVan zs orelse
           (* vissza kell lépni, ha már minden sort megpróbált *)
           z = n
        then raise Zsakutca
        else if length zs = n
        then rev zs (* megvan egy megoldás *)
        else (* folytatja a 0. sortól a következő vezér lerakásával,
              és ha elakad, visszalép a következő sorra *)
              vez 0 (z::zs) handle Zsakutca => vez (z+1) zs
      in
        (* a 0. sorral kezd: ilyenkor nem lehet ütésben *)
        vez 0 []
      end
  end

```

n vezér a sakktáblán: több megoldás előállítása visszalépéssel

```

(* vezerek1 : int -> int list list
   vezerek1 n = a feladvány összes megoldásának listája n vezér esetén
*)
fun vezerek1 n =
  let (* vez: int -> int list -> int list list
       vez z zs: az összes megoldás listája n vezér esetén
       *)
      fun vez z zs =
          if (* vissza kell lépni, ha z=0 és ütésben van, vagy ha már
              minden sort megpróbáltunk *)
             z = 0 andalso utesbenVan zs orelse z = n
          then raise Zsakutca
          else if length zs = n
             then [rev zs] (* megvan egy megoldás, listában adja vissza *)
          else (* folytatja a következő sorral, majd hozzáfűzi ... *)
              (vez (z+1) zs handle Zsakutca => []) @
              (* ... a 0. sortól kezdve a következő vezér lerakásával
                 kapott megoldásokat *)
              (vez 0 (z::zs) handle Zsakutca => [])

      in
          (* a 0. sorral kezd: ilyenkor nem lehet ütésben *)
          vez 0 []
      end
  end

```

n vezér a sakktáblán: több megoldás előállítása listák listájával

Az előző megoldás sémája sokszor használható, de ebben az egyszerű esetben felesleges: a kivétel jelzése helyett üres listát adhatunk eredményül, hiszen a kivételkezelők is csak ezt teszik.

```
(* vezerek2 : int -> int list list
   vezerek2 n = a feladvány összes megoldásának listája n vezér esetén
*)
fun vezerek2 n =
  let (* vez: int -> int list -> int list list
       vez z zs: az összes megoldás listája n vezér esetén
     *)
    fun vez z zs =
      if z = 0 andalso utesbenVan zs orelse z = n
      then []
      else if length zs = n
      then [rev zs]
      else vez (z+1) zs @ vez 0 (z::zs)
  in
    vez 0 []
  end
```

n vezér a sakktáblán: több megoldás előállítása listák listájával (folyt.)

Akkumulátor alkalmazásával:

```

(* vezerek3 : int -> int list list
   vezerek3 n = a feladvány összes megoldásának listája
                 n vezér esetén
*)
fun vezerek3 n =
  let (* vez: int -> int list -> int list list -> int list list
       vez z zs: az összes megoldás listája n vezér esetén
   *)
      fun vez z zs zss =
          if z = 0 andalso utesbenVan zs orelse z = n
          then zss
          else if length zs = n
              then rev zs :: zss
              else vez 0 (z::zs) (vez (z+1) zs zss)
      in
        vez 0 [] []
      end
  end

```